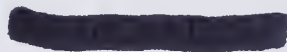


UNIVERSITY OF
ILLINOIS LIBRARY
AT URBANA-CHAMPAIGN





Digitized by the Internet Archive
in 2013

<http://archive.org/details/cleopatracodegen740halb>

570.57
Eller

UIUCDCS-R-76-740

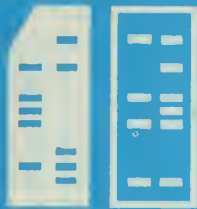
no. 740

CLEOPATRA CODE GENERATOR USER'S GUIDE

by

John David Halbur

January, 1976



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. UIUCDCS-R-76-740

CLEOPATRA CODE GENERATOR USER'S GUIDE

by

John David Halbur

1976

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

TABLE OF CONTENTS

	Page
1. THE LANGUAGE IMPLEMENTED.....	1
1.1. Introduction to this Report	1
1.2. Summary of Major Restrictions	1
1.3. Identifiers and Constants	2
1.4. Configurations	4
1.5. Types	5
1.6. Blocks	6
1.7. Expressions	7
1.8. Statements	8
1.9. Implemented Keywords	8
1.10. System Supplied Operators	8
1.11. Job Control Language	13
1.12. Subroutine Linkage	16
1.13. Error Messages	18
2. INPUT Specifications	19
2.1. SYMBOL Table Entries	19
2.2. Form of the Intermediate Text	23
2.3. Intermediate Text Transmission	23
LIST OF REFERENCES	27
APPENDIX A - SPECIFICATIONS FOR THE INTERMEDIATE TEXT	28
APPENDIX B - SUMMARY OF PRODUCTIONS IMPLEMENTED	32

1. THE LANGUAGE IMPLEMENTED

1.1 Introduction to This Report

This report consists of two parts. The first part details the language accepted by the CLEOPATRA code generator and is intended for use by CLEOPATRA programmers using the compiler. The second part describes what information should be in the symbol table and is intended for use by a compiler writer interfacing with this code generator.

Part 1 is intended as a companion to Report UIUCDCS-R-74-646 CLEOPATRA[2] which gives the specifications for the entire language. The terminology and notations used here follow that used in the earlier report. A knowledge of the information in Report 646 is essential to understanding most of what is described here.

1.2 Summary of Major Restrictions

This section describes in general some of the major restrictions of this implementation. More details appear in later sections classified by language feature.

The six major control structures are implemented. These control structures are: local and global data blocks, local and global structure blocks, and procedure and operator routine blocks. The system-supplied types are BIT, INTEGER, LONG_INTEGER, POINTER, and CHARACTER. User-defined types are allowed with some restrictions. REAL and DECIMAL types are not implemented.

There are three major restrictions on user-defined types. One is that they may not be nested. A user-defined type may not contain another user-defined type. Arrays are not allowed as part of a user-defined type but arrays of user defined types are permitted. The last restriction is that identifiers

in a user-defined type data block may not have the attributes DEFER, CONSTANT, or SHARE.

Only one storage scheme is implemented for arrays. The keywords RIGHT and VECTOR may be used where appropriate. Cross-sectioning of arrays is not implemented and all formal parameter arrays must be declared with **: bounds. Arrays may not have variable bounds and character identifiers may not have variable lengths.

The CONSTANT attribute is implemented for the basic types, but only for scalars. The INIT attribute is implemented for scalar variables only.

Deferred storage is implemented, but part of a user-defined type may not be deferred.

Procedures and operators can only return the five basic types and only as scalars. No implied type conversion is allowed anywhere in this implementation.

All of the statements in the language are implemented with a few restrictions but the features oriented toward operating systems are not implemented (these include the material in Chapters 8 and 9 of the Report 646). Data-groups are not implemented.

1.3 Identifiers and Constants

This implementation is based on the assumption that the source program will be on punched cards. However, the actual source is transparent to the code generation and some limitations on syntax placed by the analysis phase may not be included here.

All keywords are reserved. A list of keywords appears in Section 1.8. The end-of-source-record taken that delimits a comment beginning with an exclamation point is the boundary of a card.

Identifiers must be 31 characters or fewer. For configuration names the maximum length is seven because they are used as external names. If a configuration name is longer than seven characters, the first seven will be used, which could cause ambiguity.

Constants may be integer or long-integer decimal, bit, or literal string. Octal is not implemented but decimal, hexadecimal, and binary representation of integers is allowed. The minimum and maximum values for integers are given in Table 1.3.1.

Literal character strings may be at most 256 characters long.

	INTEGER		LONG_INTEGER	
	MAXIMUM	MINIMUM	MAXIMUM	MINIMUM
HEX	X.7FFF	X.8000	F.X.7FFFFFFF	F.X.80000000
DECIMAL	32767	-32768	F.2147483647	F.-2147483648

Table 1.3.1 Range of Integer Values

The system-supplied values are TRUE and FALSE for bit values, NIL for any basic type, LARGE and SMALL for integer types, and FIRST and LAST for character strings. The values assigned to these names are given in Table 1.3.2. FIRST and LAST are character strings of length one, only. They are shown in the table as their hexadecimal EBCDIC representation.

The total number of identifiers and literals for one compilation is limited to 1500.

	INTEGER	LONG-INTEGER	BIT	CHARACTER	POINTER
LARGE	32767	F.2147483647			
SMALL	-32768	F.-2147483648			
NIL	0	F.0	S.0	C.	0
TRUE			S.1		
FALSE			S.0		
FIRST				X.00	
LAST				X.FF	

Table 1.3.2 System-Supplied Values

1.4 Configurations

Configuration names may be at most seven characters. The same name may not be used as a configuration name twice in the same program. These names are used as external names. They can have aliases, but only the original name is external. This means that alias names may be longer than seven characters and may be redefined, or used in case the original name is redefined in another context.

One or two control sections will be set up for each configuration. The first is used for the program code and the second is used to store local static information such as CONSTANTS and subroutine addresses. The second may not be present if there are none of these.

Blocks should be presented for compilation grouped together by configuration. A routine block must immediately follow its corresponding data and structure blocks. Also a type data block must appear before an instance of that type is defined in a data block. Otherwise, the standard define-before-use rule applies.

A block in this implementation may not be compiled separately from other blocks in the same configuration. A configuration may be compiled separately if

- 1) it uses no global data, nor does it have a global data block. It therefore communicates with other configurations only through parameters;
- 2) it calls upon other configurations only if they are defined in this configuration's structure block; and
- 3) the routine block and data block contain no literals or constants needed at run-time except for the system-supplied constants.

Item 3 can be overcome by defining all literals and constants with the CONSTANT attribute in the local data block.

Associated only with the configuration at the root node of the configuration tree for one compilation may be an environment request. This is of the form "COMPILE INTO configuration-reference" which tells the compiler what nesting level the current compilation is to begin with. It need not be present for a main program.

Recursion is implemented for routine blocks. Parameters may not be specified for type definitions. The BUILT_IN attribute is not implemented and neither is SHARE.

1.5 Types

The basic types implemented are INTEGER, LONG_INTEGER, BIT, CHARACTER, and POINTER. Their capacities are the same as those given for their corresponding constants given in section 1.2. Identifiers of type CHARACTER must specify a maximum length. POINTER variables must point to either a basic type or a user-defined type.

Although conversion between types is not automatic, some degree of

conversion between INTEGER and LONG_INTEGER is allowed. In expressions they may be mixed without loss of precision for LONG_INTEGER values. Care must be taken, however, so that the correct type is supplied for parameters and operands of user-defined operators. Mixture in this case will result in a LONG_INTEGER result.

Pointer variables can have as a value either NIL or the address of an allocation of a deferred identifier. Storage is allocated through an ALLOCATE statement. An object can be freed only through the execution of a RELEASE statement. Loss of all pointers to an object result in garbage. Upon release of an object, the pointer used to reference it is unchanged. Copies of pointers to the same object are also unchanged. This implementation places all of the responsibility of management of deferred storage on the programmer.

User-defined types are implemented with several rules limiting their capabilities. They may not contain arrays, other user-defined types, or elements with the attributes DEFER, CONSTANT, SHARE, or INIT. Data groups are not implemented, but user-defined types can be used to aggregate the basic types.

1.6 Blocks

Structure blocks describe calling conventions for other configurations. It is possible in CLEOPATRA to specify whether or not a parameter can be changed by its subroutine. This is enforced by the analysis phase of the compiler. All parameters are passed by reference. This means that expressions and constants are passed by value, and identifiers and array references are passed by address.

Keyword parameters are not implemented and positional parameters may not be omitted. Implied conversions are not allowed for parameters. Arrays and user-defined types may not be returned by a routine block.

Data blocks define identifiers used by routine blocks. Some possible attributes are not implemented and others are restricted. Data-groups are not implemented. For arrays, only the storage scheme RIGHT for row-major storage is implemented. RIGHT and VECTOR are permissible attributes. BIT arrays are not implemented, but arrays for the four other basic types are implemented. Arrays may not be initialized. This implies that the attribute CONSTANT cannot apply to arrays.

The INIT attribute does apply to all of the basic types applied to scalar variables. Bit variables should only be initialized with TRUE and FALSE, not S.1 and S.0. POINTER variables may only be initialized to NIL. Expressions are not allowed by the INIT attribute except for those that can be evaluated at compile time.

The attributes ALIGNED and COMPACT may not be used. All identifiers are ALIGNED. INTEGER identifiers are aligned on half-word boundaries as are CHARACTER identifiers. LONG_INTEGER and POINTER variables are aligned on full word boundaries. Bit variables are aligned on byte boundaries unless they are part of a user-defined type. This is to allow the user to define a type bit-string.

Type data blocks define the structure of user-defined types. Identifiers within a type data block cannot have attributes DEFER, CONSTANT, SHARE, or INIT, along with all other restrictions stated above for regular identifiers.

1.7 Expressions

Most of sections 6.4 and 6.5 of Report 646 do not apply to this implementation. The COPY attribute is not implemented and neither is cross-sectioning of arrays. An array bound must be an integer constant or a constant expression. The lower bound is assumed to be one if not given. Formal

parameter arrays should be defined with bounds of **:*. The compiler supplied functions LBOUND and HBOUND are implemented. Their operands must be an unqualified array name and an integer constant.

1.8 Statements

The statements of the language are, for the most part, implemented in their entirety. The exceptions are RETURN, ALLOCATE, and DECISION. The only restriction on the RETURN statement concerns what can be returned. Only scalar, system supplied types may be returned. The ALLOCATE statement cannot have parameters or an INIT attribute. Deferred identifiers cannot have the INIT attribute in their data blocks either.

The restriction on the DECISION statement is that no more than 64 switches may be associated with one expression in the decision part of the statement.

1.9 Implemented Keywords

Table 1.9.1 lists all keywords used in this implementation along with their symbol table row numbers. All keywords are reserved words.

1.10 System Supplied Operators

Table 1.10.1 lists all system-supplied operators. Most of these are explained in report 646. They are briefly described here and any differences from report 646 are noted.

Unary arithmetic operators:

- negation

ABS absolute value

Character unary operator:

LENGTH LENGTH can have as its
operand only a character
variable and not an expression

Row#	Keyword	Row#	Keyword
1	ACTION	2	ALLOCATE
3	BEGIN	4	DECISION
5	DOWNTO	6	ELSE
7	END	8	EXIT
9	FOR	10	FROM
11	IF	12	ITERATE
13	NIL	14	OPERATOR
15	PROCEDURE	16	RELEASE
17	RETURN	18	STEP
19	THEN	20	WHEN
21	UPTO	22	WHILE
23	ADDRESS	24	ALIAS
25	B	26	BIT
27	BUILT	28	BY
29	C	30	CHARACTER
31	COMMENT	32	COMPILE
33	CONSTANT	34	DATA
35	DEFER	36	EXTENTS
37	F	38	GLOBAL
39	IN	40	INIT
41	INTEGER	42	INTO
43	LONG-INTEGER	44	POINTER
45	RETURNS	46	RIGHT
47	S	48	TO
49	VECTOR	50	TYPE
51	X	52	STRUCTURE
53	FALSE	54	FIRST
55	LARGE	56	LAST
57	SMALL	58	TRUE

Table 1.9.1 Keywords

-	unary minus
ABS	absolute value
+	binary addition
-	subtraction
*	multiplication
//	division
**	exponentiation
MOD	modular arithmetic
AND	logical and
OR	logical or
¬	logical not
==	equal
¬=	not equal
>	greater than
<	less than
>= or ¬<	greater than or equal to
<= or ¬>	less than or equal to
:=	assignment
LENGTH	character string length
->	substring
<-	
"->	index
?->	verify
"	concatenation

Table 1.10.1 System Supplied Operators

Assignment :=

this is implemented for all basic types.

Comparison operators:

<	less than
>	greater than
==	equal to
≠	not equal to
>=	greater than or equal to
<=	less than or equal to
≧	not less than
≧	not greater than

these return a bit value and are implemented for
INTEGER, LONG_INTEGER and CHARACTER. In addition for
BIT and POINTER types only == and ≠ are implemented.

integer arithmetic operators:

+	
-	
*	
**	
//	Truncated division
MOD	

a negative exponent for the **

operator results in zero

string selection: -> <-

character interrogation:

"-> INDEX

?-> VERIFY

character concatenation: "

logical operators for BIT and switches:

AND

OR

¬ unary not

Two operators are provided that act
as conversions from CHARACTER to LONG_INTEGER
and vice versa. They are unary operators.
Their names are CHAR and L_INT respectively.

The operators VERIFY, CHAR, and the comparison operators for character strings are performed at run-time by a subroutine call.

For convenience, two I/O procedures are provided. One named INPUT reads one record of eighty characters and returns that string via its single parameter. Its definition should appear in any program that wishes to reference it. That definition is

```
PROCEDURE INPUT (CHARACTER BY ADDRESS).
```

```
    RETURNS LONG_INTEGER;
```

It returns the length of the record. It reads from a file named SYSIN.

The second procedure accepts a character string and prints a variable length record. It accepts strings up to length 133. The user may give a carriage control at the beginning of the string, but must specify this through the JCL. The string to be printed should be passed as parameter. The length of the record is returned. The file used is named SYSPRINT. The definition of the routine must appear in the program if it is used. A sample is:

```
PROCEDURE OUTPUT (CHARACTER). RETURNS
```

```
    LONG_INTEGER;
```

1.11 Job Control Language

There are several parameters that may be passed to the code generator.

The LIST/NOLIST option specifies whether or not a listing of the code generated is to be printed. Likewise the MAP/NOMAP option specifies whether or not a memory map of constants and automatically allocated storage is to be printed. The DECK option specifies that an object deck is to be punched and the OBJECT parameter specifies that the object program should be passed to the next step for execution. These two are mutually exclusive. Both cannot be performed. The CARDS option specifies that the code generator's input appears on punched

cards. This can be used for debugging the code generator or the analysis phase. The corresponding option is DISK which specifies that the input is in a disk temporary data set. The default options are LIST, MAP, OBJECT, and DISK. If two of a pair of mutually exclusive parameters are present, the second is chosen.

The files needed by the code generator are STEPLIB to specify to the operating system where the program resides, SYSPRINT for printer output of the code and memory map, SYSIN for input, SYSLIN if the OBJECT parameter is in effect, and SYSPUNCH if the DECK parameter is specified. The job control language needed to generate code and execute that code are specified below in Table 1.11.1.

For execution the SYSLIB DD statement must include the data set described in Table 1.11.1. It contains compiler-supplied subroutines. It also contains the control section that forms the entry point of any CLEOPATRA environment of the stack and storage management.

//CODE	EXEC	PGM=CLEOCDE, PARM='CARDS'
//STEPLIB	DD	DSN=USER.P6416.CLEOCODE, DISP=SHR
//SYSLIN	DD	DSN=&&LINKFILE, DISP=(NEW,PASS),
//		SPACE=(TRK,(10,1)), UNIT=DISK
//SYSPRINT	DD	SYSOUT=A
//SPUNCH	DD	SYSOUT=B
//SYSIN	DD	*
//GO	EXEC	PGM=LOADER, PARM='EP=CLEOMAIN'
//SYSLIB	DD	DSN=USER.P6416.CLEOLIB, DISP=SHR
//SYSPRINT	DD	SYSOUT=A
//SYSUDUMP	DD	SYSOUT=A
//SYSLIN	DD	DSN=&&LINKFILE, DISP=(OLD,DELETE)
//SYSLOUT	DD	SYSOUT=A
//SYSIN	DD	*

Table 11.1.1 Suggested JCL To
Generate Code and execute CLEOPATRA

1.12 Subroutine Linkage

If it is necessary to write routines in assembler language to interface with CLEOPATRA, certain calling conventions should be followed. Parameters are passed through a pointer in register 1 to a list of addresses of parameters. Integer and pointer values are returned through register 0 and character and bit values are returned through the last entry in the parameter address vector. All registers should be stored in the save area pointed to by register 13. Register 14 contains the return address. To comply exactly with the CLEOPATRA linkage convention use the code in Table 1.12.1. Remember that the entry point to all CLEOPATRA programs must be the program named CLEOMAIN. The name of the main program is always changed to CLEOSTRT to begin the user's program and a control section named CLEOSTRT contains all literals and must be present.

At offset eight from the entry point should be the address of the static area associated with this routine if any. It is an external address. At offset 12 is a half-word containing the amount of automatically allocated storage needed for this routine. It is always a minimum of eighteen words for the standard save area. The next half-word at offset 14 contains the length of the save area and data areas. The difference between the total area and this area is the length of the temporary stack pointed to by register 12. If not enough room is available in currently allocated memory a memory management routine is called.

For specialized user assembler subroutines some or most of this may be omitted.

```

USING *, 15
STM 14,12, 12(13)
BC 15, 16( 0,15)
DC V(STATIC)
DC X'10001000'
ST 13, 8( 0,13)
L 11, 8( 0,15)
LH 12, 14( 0,15)
LH 2, 12( 0,15)
AL 2, 72( 0,13)
C 2, 76( 0,13)
BC 11, 70( 0,15)
L 4, 72( 0,13)
ST 13, 4( 0, 4)
ST 2, 72( 0, 4)
L 2, 76( 0,13)
ST 2, 76( 0, 4)
LR 13, 4
BC 15, 88( 0,15)
LR 2, 0
LR 10, 1
LH 0, 12( 0,15)
L 15, 0( 0, 3)
BALR 14,15
LR 0, 2
LR 1,10
AR 12,13
BALR 2, 0
USING *, 2

```

Subroutine Entry

```

L 13, 4( 0,13)
L 14, 12( 0,13)
LM 1,12, 24(13)
BCR 15,14

```

Subroutine Return

Table 1.12.1 Subroutine Linkage

1.13 Error Messages

The input to the code generator is supposed to be free of syntax errors, since the analysis phase should have corrected or deleted them. The code generator does generate a few error messages.

If an illegal token is passed in the intermediate text a message will be printed and the code generator will stop. It will also stop if the symbol table or constant table overflow.

The run time temporary stack cannot exceed 4096 bytes nor can the static control section connected to a routine control section. A message will be printed in this case, but code generation will continue because the code generated may still be correct. To correct an error caused by this overflow split the subroutine in question and try again.

A message will be printed if a compiler parameter is illegal, but code generation will continue. A statement might be too long for the code generator to handle. In this case code generation will terminate.

2. INPUT Specifications

2.1 SYMBOL Table Entries

Described here are the requirements for all possible entries in the symbol table:

All items have a type. The TYPE field in the table should be set to the binary encoding of the following assignments, Long integer 1, Integer 15, Pointer 7, Character 2, Bit 8, User 9, Label 13. The type field should be set to "user" only for identifiers which have a type defined elsewhere by the user.

For local data the LOCAL should set the DEFER bit to true. All formal parameters should have the bit FORMAL set to true in their row.

For numeric and character literals the LITERAL bit should be set. For INTEGER and BIT values the actual value should appear in ATR2. LONG_INTEGER values should be divided and stored in ATR1 and ATR2. Character literals should have their lengths stored in ATR1 and their value should be stored in the Constant Table. ATR2 should point to the first character of this entry.

CONSTANT storage follows the same pattern except with the CONSTANT bit set. The value to which the constant is initialized should not appear in a row by itself unless it is used elsewhere in the program.

A POINTER identifier should include in the PTR field the bit pattern of the type it points to. If this type is user defined then ATR1 should point to the row represented by the declaration of the type. Items that are part of a user-defined type (i.e., they appear in a type data block) must have the IN-TYPE bit set. In addition, the BLKLEVEL field of the first element in the type should contain the pointer to the row represented by the definition of the type. An identifier of this type should have its ATR2 entry also point to the row

represented by the type name.

Identifiers that are initialized, except for CONSTANTS, should have the INITIAL bit set. The ATR2 entry should then point to the row of the literal to which the identifier is to be initialized.

For arrays the ARRAY bit should be set and the ATR2 entry should point to the constant table where, in entries of five characters each, should first appear the number of extents followed by the lower and upper bound for extent one and so on with the lower and upper bounds for the remaining extents.

For an alias name the ALIAS bit should be set. The ATR2 entry should point to the symbol table row for the original name. In all of the above cases, for type CHARACTER the ATR1 entry should contain the maximum length, if known.

Configuration names and operator links require two adjacent symbol table rows. The LINK bit should be set for these. For a procedure name the ENTRY bits should be set to '01'B. Unary operators should have an ENTRY value of '10'B and binary operators should have '11'B. In the first row the BLK# entry should be the block in which the identifier is defined (i.e., the block numbers of the configuration in whose structure block this name is defined). The BLKLEVEL entry should contain the nesting level of the configuration represented by that name. The ATR1 entry should contain its actual block number and the ATR2 entry should contain a pointer to the position in the constant table where the character string name of the configuration appears.

In the second row the ATR1 entry should contain the number of parameters to the configuration and ATR2 is used to point to a user-defined type if the configuration returns the type POINTER to a user-defined type.

The BLK# entry always contains the block number of the block in which

the identifier is defined. The BLKLEVEL entry should contain the nesting level of the configuration in which it is defined, except in the cases mentioned above. Block numbers and block levels always start at 1. A diagram of the symbol table declaration appears in Table 2.1.1.

```

DECLARE      1      SYMTAB(95:1594),
              2  ATTRIBUTES,
                3  TYPE          BIT(4),      /* Type of an identifier */
                3  CONSTANT      BIT(1),      /* CONSTANT attribute   */
                3  ARRAY          BIT(1),      /* Set if an array       */
                3  LITERAL        BIT(1),      /* CHAR or INT constant  */
                3  INITIAL        BIT(1),      /* Set if initialized    */
                3  LOCAL          BIT(1),      /* if local data         */
                3  DEFER          BIT(1),      /* DEFER attribute       */
                3  FORMAL         BIT(1),      /* formal parameter      */
                3  ALIAS          BIT(1),      /* if alias name         */
                3  IN_TYPE        BIT(1),      /* if part of user type  */
                3  LINK           BIT(1),      /* link item             */
                3  ENTRY          BIT(2),      /* three types of entry  */
                3  PTR            BIT(4),      /* type pointed to       */
                3  APTR           BIT(1),      /* if PTR points to ARRAY*/
                3  UNUSED         BIT(11),
              2  BLK#             FIXED BIN,
              2  BLKLEVEL         FIXED BIN,
              2  ATR1             FIXED BIN,
              2  ATR2             FIXED BIN;

```

Table 2.1.1 The Symbol Table

Global structure blocks appear in configurations that describe user defined types. The BLK# entry for all configuration names found in a global structure block should correspond to the block number of the configuration in which the type is defined. The BLKLEVEL entry should be one greater than the block level of the one in which the type is defined and not two greater.

Symbol table rows corresponding to the definition of a user defined type should have a BLK# entry of zero. User defined type names do not have a limit of 7 characters and may be up to 31 characters long.

2.2 Form of the Intermediate Text

The intermediate text is described in detail elsewhere[1]. Appendix A includes the specifications for the text. It is mainly an array of symbol table pointers. Expressions are in a hybrid postfix form with array references and procedure calls in prefix form.

All expressions must be ended with an ending symbol such as a semi-colon, and all statements must end in a semi-colon, even if they do not appear in the source program. Postfix operators must be preceded in the text by first their right operand and secondly by their left operand. This is because CLEOPATRA expressions are evaluated right to left.

Logical expressions must include pointers to other entries in the intermediate text. These point to AND and OR operators that precede the corresponding logical sub-expression. The values are actually the negative of the intermediate text position.

2.3 Intermediate Text Transmission

The intermediate text should be in a data set of variable length records if the DISK compiler option is specified. Records are of three types, symbol table rows, constant table segments, and intermediate text statements.

Records should appear in the following order for a single configuration: first symbol table rows corresponding to that configuration, then constant table entries, and last the intermediate text. This should be repeated for every configuration. Configurations defining a user-defined type do not have a routine block and so their symbol table entries should be transmitted along with the information for another type of configuration.

Symbol table rows should be written one row as one logical record. Every new row since the last transmission of symbol table rows should be sent in one group. To mark the end of symbol table records a dummy record should be written, containing all 1 bits in the UNUSED entry of the record.

Following this should be parts of the constant table. These should be variable length character strings of maximum length 100. The code generator will try to read these records until it finds one of length less than 100. Next it will try to read the intermediate text. This should be transmitted one statement at a time in an array of 50 elements. A statement that requires more than fifty entries will be too long and a compiler error message should be generated in the analysis phase.

The end of a statement is a semi-colon. Compound statements are broken up. The end of an IF statement for this purpose is its corresponding THEN. The end of a DECISION statement is the word ACTION. ELSE is transmitted along with the statement that follows it. BEGIN is transmitted separately. The end of an ITERATE statement follows the semi-colon after ITERATE or the semi-colon after the WHILE expression of one exists. The code generator will stop reading the text when the logical end of the program is found. An example is diagramed in Figure 2.2.1.

If the CARDS parameter is in effect the format of the records is different.

Symbol table rows 95 - 120

CONSTANT TABLE POS 1 - 50

IT PROC stmt
Assign stmt
Return stmt
END

Symbol table rows 121-130

CONSTANT TABLE POS 51 - 150

CONSTANT TABLE POS 151 - 160

IT PROC stmt
Iterate stmt
Return stmt
END
END

Figure 2.2.1 Text Transmission

Symbol table rows should appear as one per card. The thirty-two bits should appear in the first thirty-two columns and the four other entries in the symbol table should be in the next twenty columns, five columns each. The constant table must be on one card per configuration. Following it should be the intermediate text in GET LIST format. An entry of 99999 marks the end of a statement and stops reading.

LIST OF REFERENCES

- [1] Halbur, John D., "A Code Generator for the CLEOPATRA Language", Report UIUCDCS-R-75-739, Department of Computer Science, University of Illinois, June 1975.
- [2] Schreiner, Axel T., "CLEOPATRA Comprehensive Language for Elegant Operating System and Translator Design", Report UIUCDCS-R-74-646, Department of Computer Science, University of Illinois, May 1974.
- [3] _____, "CLEOPATRA A Proposal for Another System Implementation Language", Report UIUCDCS-R-74-654, Department of Computer Science, University of Illinois, June 1974.

APPENDIX A - SPECIFICATIONS FOR THE INTERMEDIATE TEXT

The form of the BNF used here is the same as that used by Schreiner [2]. Upper case symbols and punctuation are terminals and represent their symbol table row pointers. Lower case designates non-terminals. Other notation used is:

- | indicates a choice
- { } are used to combine a number of choices
- [] the enclosed entity may appear or be omitted
- []. the enclosed entity may appear 0 or more times
- { }. the enclosed entity must appear one or more times.

intermediate text	::=	configuration_name routine_type statements END
routine_type	::=	proc op
proc	::=	PROCEDURE configuration_name [parms];
op	::=	OPERATOR [left] name [parms] right;
name	::=	identifier
parms	::=	identifier identifier parms
left	::=	identifier
right	::=	identifier
statements	::=	statement statement statements
statement	::=	stmt; lstmt
stmt	::=	expression RETURN expression ALLOCATE identifier FOR identifier RELEASE expression EXIT [label] NIL IF expression THEN statement ELSE statement

lstmt	::=	label : cstmt label; cstmt ;
cstmt	::=	begin decision iterate
begin	::=	BEGIN statements END
decision	::=	DECISION decions ACTION actions [ELSE statement] END
decisions	::=	{switches VECTOR constant constant : expression ; identifier : expression ; }.
actions	::=	{expression : statement }.
switches	::=	identifier identifier switches
iterate	::=	[FOR identifier [FROM expression;] [DOWNTO UPTO] expression ;] [STEP expression;]]ITERATE {; WHILE expression ; } statements [WHEN expression;] END

expression	::=	operand [operand] operator NOT expression) operand expression · expression operand [operand] operator itptr
operand	::=	expression identifier array proc_call
array	::=	identifier expression [, expression].)
proc_call	::=	configuration_name expression [, expression].)
operator	::=	system_supplied op_call
op_call	::=	name expression [, expression].)
configuration_name	::=	identifier
itptr	::=	- intermediate text pointer
identifier	::=	symbol table pointer to a user- defined symbol
constant	::=	symbol table pointer to a literal constant

APPENDIX B - SUMMARY OF PRODUCTIONS IMPLEMENTED

Report
74-646 #'s

- (1.1)* letter ::= A | B | C | . . . | Z
- (1.2) delimiting-character ::= (|) | . | : |
 ; | , | ' | blank-character
- (1.3)* special-character ::= @ | # | \$ | % | &
 | * | - | + | = | " | / | > | <
 | ~ |
- (1.4) digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
- (1.5) control-character ::= backspace | ! |
 end-of-source-record
- (1.6) comment ::= COMMENT any sequence of
 characters with the exception
 of a semicolon;
- (1.7) ::= ! any sequence of characters
 with the exception of an end of
 source record end-of-source-record

* modified from Report UIUCDCS-R-646

- (2.1) identifier ::= letter [letter | digit]
- (2.2) operator ::= {special-character} . |
 identifier
- (2.3) constant ::= integer | long-integer | bit
 | literal-value
- (2.4) decimal-digit ::= digit
- (2.5) hexadecimal-digit ::= digit | A | B | C | D | E | F
- (2.7) binary-digit ::= 0 | 1
- (2.8) minus-symbol ::= -
- (2.9) decimal-string ::= [minus-symbol]
 {decimal-digit}.
- (2.10) hexadecimal-string ::= X. [minus-symbol]
 {hexadecimal-digit}.
- (2.12) binary-string ::= B. [minus-symbol]
 {binary-digit}.
- (2.13)* basic-constant ::= decimal-string |
 hexadecimal-string | binary-string
- (2.14) integer ::= basic-constant
- (2.15) long-integer ::= F.integer
- (2.17) bit ::= S.integer
- (2.28) literal-value ::= C.any sequence of
 characters upto and not including
 the first following blank character
- (2.29) system-supplied-value ::= type {LARGE |
 NIL | SMALL}
- (2.30)* system-supplied-constant ::= FALSE | FIRST |
 LAST | TRUE

- (3.1) configuration ::= type-pack | algorithm
- (3.2) type-pack ::= global-structure-block
 [structure-block][data-block]
 type-data-block
- (3.3) algorithm ::= [structure-block]
 [global-data-block] [data-block]
 routine-block
- (3.4) compilation ::= [environment request]
 {structure-block | global-structure-block |
 data-block | global-data-block |
 type-data-block | routine-block}.
- (3.5) environment-request ::= COMPILE INTO
 configuration-reference
- (3.6) configuration-reference ::=
 configuration-name [.configuration-name].

- (4.1)* `basic-ref-type ::= INTEGER |`
 `LONG_INTEGER | BIT | CHARACTER |`
 `POINTER (ref-type)`
- (4.2)* `basic-type ::= INTEGER |`
 `LONG_INTEGER | BIT |`
 `CHARACTER[(expression)] | POINTER(type)`
- (4.4)* `ref-type ::= {basic-ref-type | type-name}`
 `[integer EXTENTS]`
 `[ALIAS identifier]`
- (4.5)* `type ::= {basic-type | type-name}`
 `[array]`
 `[ALIAS identifier]`

- (5.1) structure-block ::= STRUCTURE
 configuration-name { ; link-item}.
 [;] END configuration-name [;]
- (5.2) configuration-name ::= procedure-name |
 type-name | operator-link
- (5.3)* link-item ::= TYPE type-name
 [ALIAS identifier]
 | global-link-item
- (5.4)* global-link-item ::= PROCEDURE procedure-name
 [ALIAS identifier] [ref-type-list].
 RETURNS basic-type
- (5.5)* ::= operator-link : OPERATOR
 [left-ref-types] operator
 [ALIAS identifier]
 right-ref-types RETURNS basic-type
- (5.10) ref-type-list ::= (type-formal[,
 type-formal].)
- (5.11)* type-formal ::= ref-type [BY ADDRESS]
- (5.12) left-ref-types ::= ref-type [BY ADDRESS :
 [ref-type-list] | . ref-type-list]
- (5.13) right-ref-types ::= ref-type |
 : ref-type BY ADDRESS |
 ref-type-list { . ref-type | : ref-type
 BY ADDRESS}
- (5.14) global-structure-block ::= GLOBAL STRUCTURE
 type-name { ; global-link-items}.
 [;] END type-name [;]
- (5.15) type-name ::= identifier
- (5.16) routine-block ::= PROCEDURE procedure-name
 [name-list .] { ; statements}. [;]
 END procedure-name [;]
- (5.17) procedure-name ::= identifier
- (5.18) name-list ::= (formal[,formal].)
- (5.19)* formal ::= identifier
- (5.20) procedure-call ::= procedure-name
 [parameters]

- (5.21)* parameters ::= (actual[, actual].)
- (5.22)* actual ::= expression
- (5.23) routine-block ::= operator-link : OPERATOR
 [left-names] operator right-names
 { ; statement}. [;]
 END operator-link [;]
- (5.24) operator-link ::= identifier
- (5.25) left-names ::= ref-type identifier
 [{ : | . } name-list | :]
- (5.26) right-names ::= [:] ref-type
 identifier | name-list { : | . }
 ref-type identifier
- (5.27) operator-call ::= [expression [{ : | . }
 [parameters]]] operator [[parameters]
 { : | . }] expression
- (5.29) data-block ::= DATA configuration-name
 { ; [CONSTANT | DEFER] data-group}.
 [;] END configuration-name [;]
- (5.30) global-data-block ::= GLOBAL DATA
 configuration-name { ; [CONSTANT |
 DEFER] data-group}. [;] END
 configuration-name [;]
- (5.31)* type-data-block ::= GLOBAL DATA type-name
 { ; basic-type [array]
 item[,item]. } [;] END
 type-name [;]
- (5.32)* data group ::= {basic-type | type-name}
 [array] item [,item].
- (5.34)* array ::= { RIGHT | VECTOR} (bound
 [,bound].)
- (5.35)* bound ::= { **: | [integer:] integer}
- (5.36)* item ::= identifier [INIT constant]

```
(6.1)*      expression ::= constant | system-supplied-value
              | system-supplied-constant |
              array-reference | procedure-call
              | operator-call | (expression)
```

(6.2)* array-reference ::= array-expression

(6.3)* array-expression ::= identifier

- (7.1) stmt ::= expression | NIL
- (7.2) stmt ::= RETURN expression
- (7.3) stmt ::= EXIT [label]
- (7.5)* stmt ::= ALLOCATE identifier
 FOR pointer
- (7.4) stmt ::= IF expression THEN statement
 [[;] ELSE statement]
- (7.6) stmt ::= RELEASE expression
- (7.7) lstmt ::= cstmt | label : cstmt label
- (7.8) label ::= identifier
- (7.9) cstmt ::= BEGIN statement [;statement]·[;] END
- (7.10) cstmt ::= [for-phrase] ITERATE [while-phrase]
 statement [;statement]·
 [;] [when-phrase] END
- (7.11) for-phrase ::= FOR identifier [FROM expression]
 [;] [{DOWNTO | UPTO} expression]
 [;] [STEP expression] [;]
- (7.12) while-phrase ::= WHILE expression ;
- (7.13) when-phrase ::= WHEN expression [;]
- (7.14) cstmt ::= DECISION decision
 [;decision]· [;] ACTION
 action [;action]· [;]
 [ELSE statement [;]] END
- (7.15) decision ::= switch : expression
- (7.16) decision ::= switch{,switch}· [list-layout] :
 list-index
- (7.17) switch ::= identifier
- (7.18)* list-layout ::= VECTOR (integer : integer)
- (7.19)* list-index ::= expression
- (7.20) action ::= switch-expression : statement
- (7.21) switch-expression ::= [¬] switch
 [switch-operator [¬] switch]·

BIBLIOGRAPHIC DATA SHEET		1. Report No. UIUCDCS-R-76-740	2.	3. Recipient's Accession No.
4. Title and Subtitle CLEOPATRA CODE GENERATOR USER's GUIDE			5. Report Date January 1976	
7. Author(s) John David Halbur			6.	
9. Performing Organization Name and Address Department of Computer Science University of Illinois at Urbana-Champaign Urbana, Illinois 61801			8. Performing Organization Rept. No.	
12. Sponsoring Organization Name and Address Department of Computer Science University of Illinois at Urbana-Champaign Urbana, Illinois 61801			10. Project/Task/Work Unit No.	
			11. Contract/Grant No.	
			13. Type of Report & Period Covered Technical Report	
			14.	
15. Supplementary Notes				
16. Abstracts CLEOPATRA is a general-purpose and systems implementation language in the style of ALGOL designed for computers similar to the IBM System/360. Among its concepts are extensions to the ALGOL block structure, user-defined data types and data access mechanisms, and user-defined 'generic' operators. The language is goto-free, and has a generalized decision table as its main control structure. An interrupt mechanism is proposed. This report, a companion to the technical report UIUCDCS-R-75-739, discusses the code generation phase of a first implementation. The implementation restrictions and the input to the code generator are described in detail. The report is primarily intended for the implementor of an analysis phase.				
17. Key Words and Document Analysis. 17a. Descriptors Code Generation Compilation Compilers Programming Languages Storage Allocation Intermediate text				
17b. Identifiers/Open-Ended Terms CLEOPATRA				
17c. COSATI Field/Group				
18. Availability Statement UNLIMITED		19. Security Class (This Report) UNCLASSIFIED		21. No. of Pages 40
		20. Security Class (This Page) UNCLASSIFIED		22. Price



FEB 23 1978

FEB 20 1981



UNIVERSITY OF ILLINOIS-URBANA
510.84 IL6R no. C002 no.740-745(1975
Cleopatra code generator user's guide /



3 0112 088402000